Linux ftrace investigation

Concept discrimination:

> Documentation/trace/ftrace.rst:
>
> Although ftrace is typically considered the function tracer, it is really a **framework** of several assorted tracing utilities.

The core principle/design is the same as GNU `gprof`. Say program gprof.c:

```
/* gprof.c */

int sum(int a, int b)
{
    return a + b;
}

int main()
{
    int f = 20;
    int s = 20;

    return sum(f, s);
}
```

Then,

```
$ gcc gprof.c -o gprof
$ objdump -d gprof
```

to show disassembled code of these 2 functions:

```
0000000000401106 <sum>:
  401106:       55                      push   %rbp
  401107:       48 89 e5                mov    %rsp,%rbp
  ...

000000000040111a <main>:
  40111a:       55                      push   %rbp
  40111b:       48 89 e5                mov    %rsp,%rbp
  40111e:       48 83 ec 10             sub    $0x10,%rsp
  ...
```

nothing special.

If compile it with -pg

```
$ gcc -pg gprof.c -o gprof-pg
```

then

```
$ objdump -d gprof-pg
```

will show

```
0000000000401196 <sum>:
  401196:       55                      push   %rbp
  401197:       48 89 e5                mov    %rsp,%rbp
  40119a:       48 83 ec 08             sub    $0x8,%rsp
  40119e:       e8 9d fe ff ff          callq  401040 <mcount@plt>
  ...

00000000004011b3 <main>:
  4011b3:       55                      push   %rbp
  4011b4:       48 89 e5                mov    %rsp,%rbp
  4011b7:       48 83 ec 10             sub    $0x10,%rsp
  4011bb:       e8 80 fe ff ff          callq  401040 <mcount@plt>
  ...
```

Function **mcount** is from glibc, and the profiling work is done in it.

ftrace works the same way by providing a equivalent tracer function as **mcount**, one can do any tracing work in this function except for tracing function call, even for live patching.

The idea above is what Linux ftrace depends on. Concretely, x86_64 uses following GCC flags for ftrace:

```
-pg -mrecord-mcount -mfentry
```

`-pg` is **Program Instrumentation Option** in GCC terminology, so does `-fpatchable-function-entry=N[,M]` (for ftrace on ARM64). Here is the introduction from GCC doc:

> GCC supports a number of command-line options that control adding run-time instrumentation to the code it normally generates. For example, one purpose of instrumentation is collect profiling statistics for use in finding program hot spots, code coverage analysis, or profile-guided optimizations. Another class of program instrumentation is adding run-time checking to detect programming errors like invalid pointer dereferences or out-of-bounds array accesses, as well as deliberately hostile attacks such as stack smashing or C++ vtable hijacking. There is also a general hook which can be used to implement other forms of tracing or function-level instrumentation for debug or program analysis purposes.

the other 2 flags is documented as:

> -mfentry
>
> If profiling is active (‘-pg’), put the profiling counter call before the prologue.
>
> -mrecord-mcount
>
> If profiling is active (‘-pg’), generate a __mcount_loc section that contains pointers to each profiling call. This is useful for automatically patching and out calls.

Take a look at the effect of these flags, still say gprof.c as above:

```
$ gcc -pg -mfentry -mrecord-mcount gprof.c -o gprof-fentry
$ objdump -D gprof-fentry
```

will show

```
0000000000401196 <sum>:
  401196:       e8 b5 fe ff ff          callq  401050 <__fentry__@plt>
  40119b:       55                      push   %rbp
  40119c:       48 89 e5                mov    %rsp,%rbp
  40119f:       48 83 ec 08             sub    $0x8,%rsp
  ...

00000000004011b3 <main>:
  4011b3:       e8 98 fe ff ff          callq  401050 <__fentry__@plt>
  4011b8:       55                      push   %rbp
  4011b9:       48 89 e5                mov    %rsp,%rbp
  4011bc:       48 83 ec 10             sub    $0x10,%rsp
  ...

Disassembly of section __mcount_loc:

0000000000402010 <__mcount_loc>:
  402010:       96                      xchg   %eax,%esi
  402011:       11 40 00                adc    %eax,0x0(%rax)
  402014:       00 00                   add    %al,(%rax)
  402016:       00 00                   add    %al,(%rax)
  402018:       b3 11                   mov    $0x11,%bl
  40201a:       40 00 00                add    %al,(%rax)
  40201d:       00 00                   add    %al,(%rax)
```

At a quick glance, the difference between `-pg` and `-pg -mfentry` looks quite obvious: **calling different function at different place**. And the content of *__mcount_loc* section is the address of "call fentry" in each function.

Best reading to get a general idea of ftrace framework until now.

Knowing the background, let's find out how ftrace is implemented.

Preparation work for ftrace under x86_64

This is mostly about the kbuild process.

arch/x86/Kconfig:

```
config X86
    def_bool y
    select HAVE_C_RECORDMCOUNT
    ...
    select HAVE_DYNAMIC_FTRACE
    select HAVE_DYNAMIC_FTRACE_WITH_REGS
    select HAVE_DYNAMIC_FTRACE_WITH_DIRECT_CALLS
    ...
    select HAVE_FENTRY              if X86_64 || DYNAMIC_FTRACE
    select HAVE_FTRACE_MCOUNT_RECORD
    select HAVE_FUNCTION_GRAPH_TRACER
    select HAVE_FUNCTION_TRACER
    ...
```

Shows the ftrace related features that x86 support. Whether certain feature is enabled or not depends on configuration.(Refer: kernel/trace/Kconfig)

ftrace will record the address of each "call fentry" instruction in kernel function, the instruction is generated by `-pg -mfentry` flag; recording the address is done via "-mrecord-mcount" flag on GCC 5 and above, or scripts/recordmcount.{pl,c,h} when GCC doesn't support this flag.

> **NOTE**: If CPU arch support *-mfentry*, then ftrace is default to use *-mfentry*, and this is the case for x86.

On the other hand, glibc provides "fentry" function for ordinary userspace compilation, but compiling Linux kernel doesn't involve glibc, so Linux kernel provides it by itself in **arch/x86/kernel/ftrace_{64,32}.S**:

```
SYM_FUNC_START(__fentry__)
    retq
SYM_FUNC_END(__fentry__)
EXPORT_SYMBOL(__fentry__)
```

Recall: x86 Linux kernel use "call fentry" as default, as said above. This can be verified by checking the disassembled code of kernel:

```
kernel/sched/stats.o:     file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <schedstat_stop>:
   0:   e8 00 00 00 00          callq  5 <schedstat_stop+0x5>
   5:   c3                      retq
   ...

0000000000000010 <show_schedstat>:
  10:   e8 00 00 00 00          callq  15 <show_schedstat+0x5>
  15:   41 56                   push   %r14
  ...
```

then `objdump -d vmlinux` shows:

```
ffffffff8112a870 <schedstat_stop>:
ffffffff8112a870:       e8 cb 70 8d 00          callq  ffffffff81a01940
<__fentry__>
ffffffff8112a875:       c3                      retq
                       ...
ffffffff8112a880 <show_schedstat>:
ffffffff8112a880:       e8 bb 70 8d 00          callq  ffffffff81a01940
<__fentry__>
ffffffff8112a885:       41 56                   push   %r14
                       ...
```

Now let's see how kbuild do it.

In top Makefile:

```
# The arch Makefiles can override CC_FLAGS_FTRACE. We may also append it later.
# [FNST] The general switch flag for ftrace, other flags(-mrecord-mcount,
# -mfentry) should be used with -pg together.
ifdef CONFIG_FUNCTION_TRACER
  CC_FLAGS_FTRACE := -pg
endif

ifdef CONFIG_FUNCTION_TRACER  # default Y

  ifdef CONFIG_FTRACE_MCOUNT_RECORD # default Y
    # gcc 5 supports generating the mcount tables directly.
    # [FNST]: -mrecord-mcount has the same function as recordmcount.{pl,c,h}.
    # When GCC doesn't support this flag, use recordcmount.{pl,c,h} instead.
    ifeq ($(call cc-option-yn,-mrecord-mcount),y)
      CC_FLAGS_FTRACE     += -mrecord-mcount
      export CC_USING_RECORD_MCOUNT := 1
    endif
    ...
  endif

# [FNST]: Modern GCC has flag "-mfentry", this is default for x86.
ifdef CONFIG_HAVE_FENTRY # X86 support it
  ifeq ($(call cc-option-yn, -mfentry),y)
    CC_FLAGS_FTRACE     += -mfentry
    CC_FLAGS_USING      += -DCC_USING_FENTRY
  endif
endif

export CC_FLAGS_FTRACE
KBUILD_CFLAGS   += $(CC_FLAGS_FTRACE) $(CC_FLAGS_USING)
KBUILD_AFLAGS   += $(CC_FLAGS_USING)

ifdef CONFIG_DYNAMIC_FTRACE # default Y
        ifdef CONFIG_HAVE_C_RECORDMCOUNT # x86 support it
                BUILD_C_RECORDMCOUNT := y
                export BUILD_C_RECORDMCOUNT
        endif
endif

endif # CONFIG_FUNCTION_TRACER
```

scripts/Makefile.build:

```
ifdef CONFIG_FTRACE_MCOUNT_RECORD
  ifndef CC_USING_RECORD_MCOUNT
    # compiler will not generate __mcount_loc section, use recordmcount
    # or recordmcount.pl
    # [FNST] Since modern GCC support -mrecord-mcount, means
    # CC_USING_RECORD_MCOUNT is defined. Omit this code path.
    ifdef BUILD_C_RECORDMCOUNT
      sub_cmd_record_mcount = ...
      recordmcount_source := ...
    else
      sub_cmd_record_mcount = ...
      recordmcount_source := ...
    endif # BUILD_C_RECORDMCOUNT

    cmd_record_mcount = ...
  endif # CC_USING_RECORD_MCOUNT

# [FNST] So, cmd_record_mcount is emtpy.
endif # CONFIG_FTRACE_MCOUNT_RECORD

# Built-in and composite module parts.
# [FNST]: rule to compile single .c --> .o. $(call if_changed_rule,cc_o_c) is
# expanded into rule_cc_o_c as showed below.
$(obj)/%.o: $(src)/%.c $(recordmcount_source) $(objtool_dep) FORCE
        $(call cmd,force_checksrc)
        $(call if_changed_rule,cc_o_c)

# [FNST] Many things are done as following during compiling *.c to *.o,
# besides compilation itself.
define rule_cc_o_c
        $(call cmd,checksrc)
        $(call cmd_and_fixdep,cc_o_c)
        $(call cmd,gen_ksymdeps)
        $(call cmd,checkdoc)
        $(call cmd,objtool)
        $(call cmd,modversions_c)
        $(call cmd,record_mcount)
endef
```

Anyway, now we know ftrace will have *__mcount_loc* section produced for object file, by GCC flag, or by scripts/recordmcount{pl,c,h}. Entry of this section is the address of "call fentry" instruction in each function, 8 bytes under x86_64.

How kernel tools recordmcount.{pl,c.h} create *__mcount_loc* section? Refer: head comments of scripts/recordmcount.pl.

Then when linking into vmlinux, all *__mcount_loc* sections of object files are collected into *.init.data* section of vmlinux:

1. arch/x86/kernel/vmlinux.lds.S
2. include/asm-generic/vmlinux.lds.h

```
#ifdef CONFIG_FTRACE_MCOUNT_RECORD
/*
 * The ftrace call sites are logged to a section whose name depends on the
```

```
 * compiler option used. A given kernel image will only use one, AKA
 * FTRACE_CALLSITE_SECTION. We capture all of them here to avoid header
 * dependencies for FTRACE_CALLSITE_SECTION's definition.
 *
 * Need to also make ftrace_stub_graph point to ftrace_stub
 * so that the same stub location may have different protocols
 * and not mess up with C verifiers.
 */
#define MCOUNT_REC()    . = ALIGN(8);                \
            __start_mcount_loc = .;         \
            KEEP(*(__mcount_loc))           \
            KEEP(*(__patchable_function_entries))   \
            __stop_mcount_loc = .;          \
            ftrace_stub_graph = ftrace_stub;
#else
...
#endif
```

With all these preparation work, now turn to see how ftrace code utilize the data from *__mcount_loc* section in "ftrace initialization" chapter.

Preparation work for ftrace under aarch64

Because of failure of setting up a ARM VM on x86, so for the time being, all ARM related experiments are done with cross compiler: **AArch64 GNU/Linux target (aarch64-none-linux-gnu)** of The GNU Toolchain for the Cortex-A.

arch/arm64/Kconfig has:

```
config ARM64
    ...
    select HAVE_DYNAMIC_FTRACE
    select HAVE_DYNAMIC_FTRACE_WITH_REGS \
            if $(cc-option,-fpatchable-function-entry=2)
    ...
    select HAVE_FTRACE_MCOUNT_RECORD
    select HAVE_FUNCTION_TRACER
    select HAVE_FUNCTION_GRAPH_TRACER
```

Re-analyze top Makefile & scripts/Makefile.build for arm64.

In top Makefile:

```
# The arch Makefiles can override CC_FLAGS_FTRACE. We may also append it later.
# [FNST] This is going to be override if CONFIG_DYNAMIC_FTRACE_WITH_REGS is
supported
ifdef CONFIG_FUNCTION_TRACER
  CC_FLAGS_FTRACE := -pg
endif
...
include arch/$(SRCARCH)/Makefile
...

ifdef CONFIG_FUNCTION_TRACER  # default Y for arm64
```

```
  ifdef CONFIG_FTRACE_MCOUNT_RECORD # default Y for arm64
    # [FNST]: The cross compiler doesn't support -mrecord-mcount flag.
    ifeq ($(call cc-option-yn,-mrecord-mcount),y)
      CC_FLAGS_FTRACE     += -mrecord-mcount
      export CC_USING_RECORD_MCOUNT := 1
    endif
    ...
  endif

# [FNST]: arm64 don't support HAVE_FENTRY.
ifdef CONFIG_HAVE_FENTRY
  ifeq ($(call cc-option-yn, -mfentry),y)
    CC_FLAGS_FTRACE     += -mfentry
    CC_FLAGS_USING      += -DCC_USING_FENTRY
  endif
endif

export CC_FLAGS_FTRACE
KBUILD_CFLAGS   += $(CC_FLAGS_FTRACE) $(CC_FLAGS_USING)
KBUILD_AFLAGS   += $(CC_FLAGS_USING)

ifdef CONFIG_DYNAMIC_FTRACE # default Y for arm64
    ifdef CONFIG_HAVE_C_RECORDMCOUNT # arm64 support it
        BUILD_C_RECORDMCOUNT := y
            export BUILD_C_RECORDMCOUNT
    endif
endif

endif # CONFIG_FUNCTION_TRACER
```

arch/arm64/Makefile:

```
ifeq ($(CONFIG_DYNAMIC_FTRACE_WITH_REGS),y)
    KBUILD_CPPFLAGS += -DCC_USING_PATCHABLE_FUNCTION_ENTRY
    CC_FLAGS_FTRACE := -fpatchable-function-entry=2
endif
```

It shows clearly that on arm64, CONFIG_DYNAMIC_FTRACE_WITH_REGS feature relies on GCC flag `-fpatchable-function-entry=2` (livepatch depends on this feature. Further investigation should be done). This feature is introduced by commit 3b23e4991fb66f in 2019/02. Before, CONFIG_DYNAMIC_FTRACE on arm64 only use flag `-pg`, since arm64's GCC doesn't support `-mrecord-mcount`, it can be deduced that recordmcount.{pl, c, h} is needed to create/append *__mcount_loc* section to each *.o.

scripts/Makefile.build(re-format for easy reading):

```
ifdef CONFIG_FTRACE_MCOUNT_RECORD # default Y for arm64

    ifndef CC_USING_RECORD_MCOUNT # N for arm64, so it walks this path.
        # [FNST] each object file will be processed directly by recordmcount.
        # After checking recordmcount source, suspect it is unnecessary for
        # (arm64 + CONFIG_DYNAMIC_FTRACE_WITH_REGS). Prospective patch.
        ifdef BUILD_C_RECORDMCOUNT # Y for arm64
            sub_cmd_record_mcount =
            if [ $(@) != "scripts/mod/empty.o" ]; then      \
                $(objtree)/scripts/recordmcount $(RECORDMCOUNT_FLAGS) "$(@)";  \
```

```
            fi;

            recordmcount_source := $(srctree)/scripts/recordmcount.{c, h}
        else
            sub_cmd_record_mcount = ...
            recordmcount_source := ...
        endif # BUILD_C_RECORDMCOUNT

        # [FNST] if GCC has ftrace related flags, process it with recordmcount.
        cmd_record_mcount = $(if $(findstring $(strip
$(CC_FLAGS_FTRACE)),$(_c_flags)), \
                    $(sub_cmd_record_mcount))
    endif # CC_USING_RECORD_MCOUNT

endif # CONFIG_FTRACE_MCOUNT_RECORD

# Built-in and composite module parts.
$(obj)/%.o: $(src)/%.c $(recordmcount_source) $(objtool_dep) FORCE
        $(call cmd,force_checksrc)
        $(call if_changed_rule,cc_o_c)

define rule_cc_o_c
        $(call cmd,checksrc)
        $(call cmd_and_fixdep,cc_o_c)
        $(call cmd,gen_ksymdeps)
        $(call cmd,checkdoc)
        $(call cmd,objtool)
        $(call cmd,modversions_c)
        $(call cmd,record_mcount)
```

As CONFIG_DYNAMIC_FTRACE_WITH_REGS on arm64 uses flag `-fpatchable-function-entry=2`, the following sub-section takes a look at how this flag works.

The rest process is the same as x86: collect *__patchable_function_entries* section of *.o into *.init.data* section of vmlinux, which will be utilized by ftrace initialization code.

-fpatchable-function-entry=N[,M]

As said above, arm64 support CONFIG_DYNAMIC_FTRACE_WITH_REGS with a slightly different technique background from x86, by using `-fpatchable-function-entry=N[,M]` instead of `-pg`.

GCC explains `-fpatchable-function-entry=N[,M]` as:

> Generate N NOPs right at the beginning of each function, **with the function entry point before the M th NOP**. If M is omitted, it defaults to 0 so the function entry points to the address just at the first NOP.

The meaning of `M` is a little bit tricky to understand, and its relation with `N` is also subtle. Still take gprof.c on x86 as above for example:

```
$ gcc -fpatchable-function-entry=4,0 gprof.c -o gp-pfe
```

`objdump -d pg-pfe` shows:

```
gp-pfe:     file format elf64-x86-64
...
```

```
0000000000401100 <frame_dummy>:
  401100:       f3 0f 1e fa             endbr64
  401104:       eb 8a                   jmp    401090 <register_tm_clones>

0000000000401106 <sum>:
  401106:       90                      nop
  401107:       90                      nop
  401108:       90                      nop
  401109:       90                      nop
  40110a:       55                      push   %rbp
  40110b:       48 89 e5                mov    %rsp,%rbp
  ...

000000000040111e <main>:
  40111e:       90                      nop
  40111f:       90                      nop
  401120:       90                      nop
  401121:       90                      nop
  401122:       55                      push   %rbp
  401123:       48 89 e5                mov    %rsp,%rbp
  401126:       48 83 ec 10             sub    $0x10,%rsp
  ...
```

nothing special, just as expected. Then try:

```
$ gcc -fpatchable-function-entry=4,2 gprof.c -o gp-pfe
```

and `objdump -d pg-pfe` shows the **subtle difference**:

```
gp-pfe:     file format elf64-x86-64

0000000000401100 <frame_dummy>:
  401100:       f3 0f 1e fa             endbr64
  401104:       eb 8a                   jmp    401090 <register_tm_clones>
  401106:       90                      nop
  401107:       90                      nop

0000000000401108 <sum>:
  401108:       90                      nop
  401109:       90                      nop
  40110a:       55                      push   %rbp
  40110b:       48 89 e5                mov    %rsp,%rbp
  ...
  40111d:       c3                      retq
  40111e:       90                      nop
  40111f:       90                      nop

0000000000401120 <main>:
  401120:       90                      nop
  401121:       90                      nop
  401122:       55                      push   %rbp
  401123:       48 89 e5                mov    %rsp,%rbp
  401126:       48 83 ec 10             sub    $0x10,%rsp
  ...
```

> Tip: M should be <= N, or else compilation error happens.

In terms of section, usage of `-fpatchable-function-entry=N[,M]` results in *__patchable_function_entries* section in the binary, check it out via `objdump -h pg-pfe`. What is the content in this section?

```
$ objdump -d -j __patchable_function_entries gp-pfe
```

shows:

```
gp-pfe:     file format elf64-x86-64

Disassembly of section __patchable_function_entries:

000000000040401c <__TMC_END__-0x4>:
  40401c:   06                      (bad)
  40401d:   11 40 00                adc    %eax,0x0(%rax)

0000000000404020 <__TMC_END__>:
  404020:   00 00 00 00 1e 11 40 00 00 00 00 00                 ......@.....
```

In human read format:

```
    06 11 40 00 00 00 00 00
    1e 11 40 00 00 00 00 00
```

As x86 is little-endian, convert them to real numbers:

```
    00 00 00 00 00 40 11 06
    00 00 00 00 00 40 11 1e
```

Check these number out by comparing with output of `objdump -d pg-pfe` above(**-fpatchable-function-entry=4,2** version), will see they are the addresses of 1st NOP instruction. Just as GCC explains:

> For run-time identification, the starting addresses of these areas, which correspond to their respective function entries minus M, are additionally collected in the __patchable_function_entries section of the resulting binary.

So if kernel use `-fpatchable-function-entry=2`, the address of 1st NOP is also the function entry address, which is recorded in section *__patchable_function_entries*. Now Let's verify this verdict on arm64.

Compile with `aarch64-noaarch64-none-linux-gnu-gcc -fpatchable-function-entry=2 gprof.c -o gp`, and check the disassembly with `aarch64-noaarch64-none-linux-gnu-objdump -d gp`:

```
gp:     file format elf64-littleaarch64

  ...
000000000040055c <sum>:
  40055c:       d503201f        nop
  400560:       d503201f        nop
  400564:       d10043ff        sub     sp, sp, #0x10
  ...

0000000000400584 <main>:
```

```
  400584:       d503201f        nop
  400588:       d503201f        nop
  40058c:       a9be7bfd        stp     x29, x30, [sp, #-32]!
  ...

Disassembly of section __patchable_function_entries:

0000000000411028 <__TMC_END__>:
  411028:       0040055c        .inst   0x0040055c ; undefined
  41102c:       00000000        .inst   0x00000000 ; undefined
  411030:       00400584        .inst   0x00400584 ; undefined
  411034:       00000000        .inst   0x00000000 ; undefined
```

Bingo~

ftrace initialization

Its initialization has several phase:

1. start_kernel --> ftrace_init
2. start_kernel --> early_trace_init
3. start_kernel --> arch_call_rest_init --...--> do_initcalls --> tracer_init_tracefs

ftrace_init

What does ftrace initialization do? A brief introduction omitting code details is as following.

First, ftrace has *struct dyn_ftrace* recording each entry of *__mcount_loc* section, they are 1:1 mapped. Allocate exact memory pages for all *struct dyn_ftrace* records, and initialize them with the addresses from *__mcount_loc* section, as simple as struct dyn_ftrace.ip = address.

```
struct dyn_ftrace {
    unsigned long       ip; /* address of mcount call-site */
    unsigned long       flags;
    struct dyn_arch_ftrace  arch;
};
```

All *struct dyn_ftrace* are managed via *struct ftrace_page*, stored in *struct ftrace_page.records*. In case one continuous memory allocation doesn't suffice, one or more *struct ftrace_page* involved, single-linked by its *struct ftrace_page.next*.

```
struct ftrace_page {
    struct ftrace_page  *next;
    struct dyn_ftrace   *records;
    int         index;
    int         size;
};
```

In other words: first step is to construct an initial database that consists of all the functions ftrace managed, each function is identified by its address. Note: the address of "call fentry" equals its function address, refer the output of `objdump -d vmlinux` above.

Next step is to replace all `call fentry` with NOP instructions through *ftrace_update_code* --> *ftrace_nop_initialize*. Intel has different NOP instruction implementations, i.e., a NOP instruction could be one byte or multi-byte long, as Intel software developer manual 2 shows:

NOP—No Operation

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
NP 90	NOP	ZO	Valid	Valid	One byte no-operation instruction.
NP 0F 1F /0	NOP r/m16	M	Valid	Valid	Multi-byte no-operation instruction.
NP 0F 1F /0	NOP r/m32	M	Valid	Valid	Multi-byte no-operation instruction.

Table 4-12. Recommended Multi-Byte Sequence of NOP Instruction

Length	Assembly	Byte Sequence
2 bytes	66 NOP	66 90H
3 bytes	NOP DWORD ptr [EAX]	0F 1F 00H
4 bytes	NOP DWORD ptr [EAX + 00H]	0F 1F 40 00H
5 bytes	NOP DWORD ptr [EAX + EAX*1 + 00H]	0F 1F 44 00 00H
6 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00H]	66 0F 1F 44 00 00H
7 bytes	NOP DWORD ptr [EAX + 00000000H]	0F 1F 80 00 00 00 00H
8 bytes	NOP DWORD ptr [EAX + EAX*1 + 00000000H]	0F 1F 84 00 00 00 00 00H
9 bytes	66 NOP DWORD ptr [EAX + EAX*1 + 00000000H]	66 0F 1F 84 00 00 00 00 00H

Tip: `call fentry` instruction in Linux kernel as shown above, is 5-byte long, so, of course ftrace(x86) will choose the 5 bytes NOP. NOP instruction definition code is in arch/x86/kernel/alternative.c:

```
/* Initialize these to a safe default */
#ifdef CONFIG_X86_64
const unsigned char * const *ideal_nops = p6_nops;
#else
const unsigned char * const *ideal_nops = intel_nops;
#endif

#ifdef P6_NOP1
static const unsigned char p6nops[] =
{
    P6_NOP1,
    P6_NOP2,
    P6_NOP3,
    P6_NOP4,
    P6_NOP5,
    P6_NOP6,
    P6_NOP7,
    P6_NOP8,
    P6_NOP5_ATOMIC
};
static const unsigned char * const p6_nops[ASM_NOP_MAX+2] =
{
    NULL,
    p6nops,
    p6nops + 1,
    p6nops + 1 + 2,
    p6nops + 1 + 2 + 3,
    p6nops + 1 + 2 + 3 + 4,
    p6nops + 1 + 2 + 3 + 4 + 5,
    p6nops + 1 + 2 + 3 + 4 + 5 + 6,
    p6nops + 1 + 2 + 3 + 4 + 5 + 6 + 7,
    p6nops + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8,
};
#endif
```

And ftrace on x86 use the 5-byte NOP:

```
static const unsigned char *ftrace_nop_replace(void)
{
    return ideal_nops[NOP_ATOMIC5];
}

#define NOP_ATOMIC5 (ASM_NOP_MAX+1) /* Entry for the 5-byte atomic NOP */
```

At last, initialize ftrace filters according to kernel parameters in *set_ftrace_early_filters*, take the fundamental 2 of them for example:

> ftrace_filter=[function-list]
>
> [FTRACE] Limit the functions traced by the function tracer at boot up. function-list is a comma separated list of functions. This list can be changed at run time by the set_ftrace_filter file in the debugfs tracing directory.
>
> ftrace_notrace=[function-list]
>
> [FTRACE] Do not trace the functions specified in function-list. This list can be changed at run time by the set_ftrace_notrace file in the debugfs tracing directory.

Under CONFIG_DYNAMIC_FTRACE, the functions in these [function-list] is managed via hash bucket.

```
struct ftrace_hash {
    unsigned long       size_bits;
    struct hlist_head   *buckets;
    unsigned long       count;
    unsigned long       flags;
    struct rcu_head     rcu;
};
```

The element in hash bucket is:

```
struct ftrace_func_entry {
    struct hlist_node hlist;
    unsigned long ip;
    unsigned long direct; /* for direct lookup only */
};
```

The hash function *ftrace_hash_key* take integer form of function address as input, hash value returned as the index into hash bucket.

ftrace usage

Reference

Welcome to ftrace & the Start of Your Journey to Understanding the Linux Kernel! by ftrace author.